How to Write Go Code

Introduction

This document demonstrates the development of a simple Go package and introduces the go tool, the standard way to fetch, build, and install Go packages and commands.

The `go` tool requires you to organize your code in a specific way. Please read this document carefully. It explains the simplest way to get up and running with your Go installation.

A similar explanation is available as a screencast.

Code organization

Workspaces

The `go` tool is designed to work with open source code maintained in public repositories. Although you don't need to publish your code, the model for how the environment is set up works the same whether you do or not.

Go code must be kept inside a *workspace*. A workspace is a directory hierarchy with three directories at its root:

- `src` contains Go source files organized into packages (one package per directory),
- `pkg` contains package objects, and
- `bin` contains executable commands.

The `go` tool builds source packages and installs the resulting binaries to the `pkg` and `bin` directories.

The `src` subdirectory typically contains multiple version control repositories (such as for Git or

Mercurial) that track the development of one or more source packages.

To give you an idea of how a workspace looks in practice, here's an example:

```
bin/
    hello                          # command executable
    outyet                         # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a           # package object
src/
    github.com/golang/example/
        .git/                      # Git repository metadata
        hello/
            hello.go               # command source
        outyet/
            main.go                # command source
            main_test.go           # test source
        stringutil/
            reverse.go             # package source
            reverse_test.go        # test source
```

This workspace contains one repository (`example`) comprising two commands (`hello` and `outyet`) and one library (`stringutil`).

A typical workspace would contain many source repositories containing many packages and commands. Most Go programmers keep *all* their Go source code and dependencies in a single workspace.

Commands and libraries are built from different kinds of source packages. We will discuss the distinction later.

The `GOPATH` environment variable

The `GOPATH` environment variable specifies the location of your workspace. It is likely the only environment variable you'll need to set when developing Go code.

To get started, create a workspace directory and set `GOPATH` accordingly. Your workspace can be located wherever you like, but we'll use `$HOME/work` in this document. Note that this must **not** be the same path as your Go installation. (Another common setup is to set `GOPATH=$HOME`.)

```
$ mkdir $HOME/work
$ export GOPATH=$HOME/work
```

For convenience, add the workspace's `bin` subdirectory to your `PATH`:

```
$ export PATH=$PATH:$GOPATH/bin
```

To learn more about setting up the `GOPATH` environment variable, please see `go help gopath`

Package paths

The packages from the standard library are given short paths such as "`fmt`" and "`net/http`". For your own packages, you must choose a base path that is unlikely to collide with future additions to the standard library or other external libraries.

If you keep your code in a source repository somewhere, then you should use the root of that source repository as your base path. For instance, if you have a GitHub account at `github.com/user`, that should be your base path.

Note that you don't need to publish your code to a remote repository before you can build it. It's just a good habit to organize your code as if you will publish it someday. In practice you can choose any arbitrary path name, as long as it is unique to the standard library and greater Go ecosystem.

We'll use `github.com/user` as our base path. Create a directory inside your workspace in which to keep source code:

```
$ mkdir -p $GOPATH/src/github.com/user
```

Your first program

To compile and run a simple program, first choose a package path (we'll use `github.com/user/hello`) and create a corresponding package directory inside your workspace:

```
$ mkdir $GOPATH/src/github.com/user/hello
```

Next, create a file named `hello.go` inside that directory, containing the following Go code.

```
package main

import "fmt"

func main() {
        fmt.Printf("Hello, world.\n")
}
```

Now you can build and install that program with the `go` tool:

```
$ go install github.com/user/hello
```

Note that you can run this command from anywhere on your system. The `go` tool finds the source code by looking for the `github.com/user/hello` package inside the workspace specified by `GOPATH`.

You can also omit the package path if you run `go install` from the package directory:

```
$ cd $GOPATH/src/github.com/user/hello
$ go install
```

This command builds the `hello` command, producing an executable binary. It then installs that binary to the workspace's `bin` directory as `hello` (or, under Windows, `hello.exe`). In our

example, that will be `$GOPATH/bin/hello`, which is `$HOME/work/bin/hello`.

The `go` tool will only print output when an error occurs, so if these commands produce no output they have executed successfully.

You can now run the program by typing its full path at the command line:

```
$ $GOPATH/bin/hello
Hello, world.
```

Or, as you have added `$GOPATH/bin` to your `PATH`, just type the binary name:

```
$ hello
Hello, world.
```

If you're using a source control system, now would be a good time to initialize a repository, add the files, and commit your first change. Again, this step is optional: you do not need to use source control to write Go code.

```
$ cd $GOPATH/src/github.com/user/hello
$ git init
Initialized empty Git repository in
/home/user/work/src/github.com/user/hello/.git/
$ git add hello.go
$ git commit -m "initial commit"
[master (root-commit) 0b4507d] initial commit
 1 file changed, 1 insertion(+)
  create mode 100644 hello.go
```

Pushing the code to a remote repository is left as an exercise for the reader.

Your first library

Let's write a library and use it from the `hello` program.

Again, the first step is to choose a package path (we'll use `github.com/user/stringutil`) and create the package directory:

```
$ mkdir $GOPATH/src/github.com/user/stringutil
```

Next, create a file named `reverse.go` in that directory with the following contents.

```
// Package stringutil contains utility functions for working with strings.
package stringutil

// Reverse returns its argument string reversed rune-wise left to right.
func Reverse(s string) string {
        r := []rune(s)
        for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
                r[i], r[j] = r[j], r[i]
        }
        return string(r)
```

```
}
```

Now, test that the package compiles with `go build`:

```
$ go build github.com/user/stringutil
```

Or, if you are working in the package's source directory, just:

```
$ go build
```

This won't produce an output file. To do that, you must use `go install`, which places the package object inside the `pkg` directory of the workspace.

After confirming that the `stringutil` package builds, modify your original `hello.go` (which is in `$GOPATH/src/github.com/user/hello`) to use it:

```
package main

import (
        "fmt"

        "github.com/user/stringutil"
)

func main() {
        fmt.Printf(stringutil.Reverse("!oG ,olleH"))
}
```

Whenever the `go` tool installs a package or binary, it also installs whatever dependencies it has. So when you install the `hello` program

```
$ go install github.com/user/hello
```

the `stringutil` package will be installed as well, automatically.

Running the new version of the program, you should see a new, reversed message:

```
$ hello
Hello, Go!
```

After the steps above, your workspace should look like this:

```
bin/
    hello                 # command executable
pkg/
    linux_amd64/          # this will reflect your OS and architecture
        github.com/user/
            stringutil.a  # package object
src/
    github.com/user/
        hello/
            hello.go      # command source
```

```
        stringutil/
            reverse.go    # package source
```

Note that `go install` placed the `stringutil.a` object in a directory inside `pkg/linux_amd64` that mirrors its source directory. This is so that future invocations of the `go` tool can find the package object and avoid recompiling the package unnecessarily. The `linux_amd64` part is there to aid in cross-compilation, and will reflect the operating system and architecture of your system.

Go command executables are statically linked; the package objects need not be present to run Go programs.

Package names

The first statement in a Go source file must be

```
package name
```

where *name* is the package's default name for imports. (All files in a package must use the same *name*.)

Go's convention is that the package name is the last element of the import path: the package imported as "`crypto/rot13`" should be named `rot13`.

Executable commands must always use `package main`.

There is no requirement that package names be unique across all packages linked into a single binary, only that the import paths (their full file names) be unique.

See Effective Go to learn more about Go's naming conventions.

Testing

Go has a lightweight test framework composed of the `go test` command and the `testing` package.

You write a test by creating a file with a name ending in `_test.go` that contains functions named `TestXXX` with signature `func (t *testing.T)`. The test framework runs each such function; if the function calls a failure function such as `t.Error` or `t.Fail`, the test is considered to have failed.

Add a test to the `stringutil` package by creating the file `$GOPATH/src/github.com/user/stringutil/reverse_test.go` containing the following Go code.

```
package stringutil

import "testing"

func TestReverse(t *testing.T) {
        cases := []struct {
                in, want string
        }{
```

```
                {"Hello, world", "dlrow ,olleH"},
                {"Hello, 世界", "界世 ,olleH"},
                {"", ""},
        }
        for _, c := range cases {
                got := Reverse(c.in)
                if got != c.want {
                        t.Errorf("Reverse(%q) == %q, want %q", c.in, got,
c.want)
                }
        }
}
```

Then run the test with `go test`:

```
$ go test github.com/user/stringutil
ok      github.com/user/stringutil 0.165s
```

As always, if you are running the `go` tool from the package directory, you can omit the package path:

```
$ go test
ok      github.com/user/stringutil 0.165s
```

Run `go help test` and see the testing package documentation for more detail.

Remote packages

An import path can describe how to obtain the package source code using a revision control system such as Git or Mercurial. The `go` tool uses this property to automatically fetch packages from remote repositories. For instance, the examples described in this document are also kept in a Git repository hosted at GitHub `github.com/golang/example`. If you include the repository URL in the package's import path, `go get` will fetch, build, and install it automatically:

```
$ go get github.com/golang/example/hello
$ $GOPATH/bin/hello
Hello, Go examples!
```

If the specified package is not present in a workspace, `go get` will place it inside the first workspace specified by `GOPATH`. (If the package does already exist, `go get` skips the remote fetch and behaves the same as `go install`.)

After issuing the above `go get` command, the workspace directory tree should now look like this:

```
bin/
    hello                           # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
```

```
            stringutil.a            # package object
        github.com/user/
            stringutil.a            # package object
src/
    github.com/golang/example/
        .git/                       # Git repository metadata
        hello/
            hello.go                # command source
        stringutil/
            reverse.go              # package source
            reverse_test.go         # test source
    github.com/user/
        hello/
            hello.go                # command source
        stringutil/
            reverse.go              # package source
            reverse_test.go         # test source
```

The `hello` command hosted at GitHub depends on the `stringutil` package within the same repository. The imports in `hello.go` file use the same import path convention, so the `go get` command is able to locate and install the dependent package, too.

```
import "github.com/golang/example/stringutil"
```

This convention is the easiest way to make your Go packages available for others to use. The Go Wiki and godoc.org provide lists of external Go projects.

For more information on using remote repositories with the `go` tool, see `go help importpath`.

What's next

Subscribe to the golang-announce mailing list to be notified when a new stable version of Go is released.

See Effective Go for tips on writing clear, idiomatic Go code.

Take A Tour of Go to learn the language proper.

Visit the documentation page for a set of in-depth articles about the Go language and its libraries and tools.

Getting help

For real-time help, ask the helpful gophers in `#go-nuts` on the Freenode IRC server.

The official mailing list for discussion of the Go language is Go Nuts.

Report bugs using the Go issue tracker.